Evolving Algebras and Partial Evaluation*

Yuri Gurevich† and James K. Huggins†

February 1, 2008

Keyword Codes: D.2.2; D.2.m; F.3.2
Keywords: Software Engineering, Tools and Techniques;
Software Engineering, Miscellaneous;
Logics and Meanings of Programs, Semantics of Programming Languages

Abstract

We describe an automated partial evaluator for evolving algebras implemented at the University of Michigan.

1 Introduction to Sequential Evolving Algebras

A fuller discussion of evolving algebras (or *ealgebras*) can be found in [1]; to make this paper self-contained, we recall briefly the main concepts.

A sequential ealgebra $\mathcal{A}$ is an abstract machine. The *signature* of $\mathcal{A}$ is a (finite) collection of function names, each of a fixed arity. A state of $\mathcal{A}$ is a set, the *superuniverse*, together with interpretations of the function names in the signature. These interpretations are called *basic functions* of the state. The superuniverse does not change as $\mathcal{A}$ evolves; the basic functions may.

Formally, a basic function of arity r (*i.e.* the interpretation of a function name of arity r) is an r-ary operation on the superuniverse. (We often use basic functions with $r = 0$; such basic functions will be called *distinguished elements*.) But functions naturally arising in applications may be defined only on a part of the superuniverse. Such partial functions are represented by total functions in the following manner.

The superuniverse contains distinct elements *true, false, undef* which allow us to deal with relations (viewed as binary functions with values *true* or *false*) and partial functions (where $f(\overline{a}) = \mathit{undef}$ means f is undefined at the tuple $\overline{a}$). These three elements are *logical constants*. Their names do not appear in the signature; this is similar to the situation in first-order logic with equality where equality is a logical constant and the sign of equality does not appear in the signature. In fact, we use equality as a logical constant as well.

A *universe* U is a special type of basic function: a unary relation usually identified with the set $\{x : U(x)\}$. The universe $\mathit{Bool} = \{\mathit{true}, \mathit{false}\}$ is another logical constant. When we speak about a function f from a universe U to a universe V, we mean that formally f is a unary operation on the superuniverse such that $f(a) \in V$ for all $a \in U$ and $f(a) = \mathit{undef}$ otherwise. We use self-explanatory notations like $f : U \to V$, $f : U_1 \times U_2 \to V$, and $f : V$. The last means that the distinguished element f belongs to V.

In principle, a program of $\mathcal{A}$ is a finite collection of transition rules of the form

if t_0 **then** $f(t_1, \ldots, t_r) := t_{r+1}$ **endif** *(1)*

*In *Proceedings of IFIP Congress 94 – Volume 1*, eds. B. Pehrson and I. Simon, Elsevier, 1994.

†Partially supported by ONR grant N00014-91-J-1861 and NSF grant CCR-92-04742. EECS Department, University of Michigan, Ann Arbor, MI, 48109-2122, USA. gurevich@umich.edu, huggins@umich.edu

where t_0, $f(t_1, \ldots, t_r)$, and t_{r+1} are closed terms (*i.e.* terms containing no free variables) in the signature of $\mathcal{A}$. An example of such a term is $g(h_1, h_2)$ where g is binary and h_1 and h_2 are zero-ary. The meaning of the rule shown above is this: Evaluate all the terms t_i in the given state; if t_0 evaluates to *true* then change the value of the basic function f at the value of the tuple $(t_1, .., t_r)$ to the value of t_{r+1}, otherwise do nothing.

In fact, rules are defined in a slightly more liberal way; if k is a natural number, $b_0, \ldots, b_k$ are terms and $C_0, \ldots, C_k$ are sets of rules then the following is a rule:

if b_0 **then** C_0
elseif b_1 **then** C_1
$\vdots$ *(2)*
elseif b_k **then** C_k
endif

In the case that $b_k = true$, the last line may be abbreviated by "**else** C_k".

Since the C_i are sets of rules, nested transition rules are allowed (and occur frequently).

A program is a set of rules. It is easy to transform a program to an equivalent program comprising only rules of the stricter form (1). We use rules of the more liberal form (2), as well as macros (textual abbreviations), for brevity.

How does $\mathcal{A}$ evolve from one state to another? In a given state, the demon (or interpreter) evaluates all the relevant terms and then makes all the necessary updates. If several updates contradict each other (trying to assign different values to the same basic function at the same place), then the demon chooses nondeterministically one of those updates to execute.

We call a function (name) f *dynamic* if an assignment of the form $f(t_1, \ldots, t_r) := t_0$ appears anywhere in the transition rules. Functions which are not dynamic are called *static*. To allow our algebras to interact conveniently with the outside world, we also make use of *external* functions within our algebra. External functions are syntactically static (that is, never changed by rules), but have their values determined by a dynamic oracle. Thus, an external function may have different values for the same arguments as the algebra evolves.

1.1 Why Partial Evaluation and Evolving Algebras?

One of the main application areas of ealgebras has been programming language semantics. One may view an ealgebra A for a language L as an abstract machine which acts as an interpreter for L. With an L-program p as an input, A gives semantics for p. However, A may be large; for many programs, an ealgebra tailored directly to p is clearer than A. Partial evaluation provides an automated means for tailoring an ealgebra for an L-program p by specializing an ealgebra interpreter for L with respect to p. These tailored ealgebras may not be as good as hand-tailored ones, but they may provide a useful beginning for tailored ealgebras. Of course, this is only one use of a partial evaluator for ealgebras.

2 Partial Evaluation Techniques

Suppose one has a program p and knows a portion of its input ahead of time. Can one take advantage of this information to transform p into a more efficient program? *Partial evaluation* is the process of transforming such a program p into a program p' which, when supplied with the remainder of p's input, has the same behavior as p.

Our partial evaluator follows the "mix" methodology (described in more detail in [3]) and has three phases: binding-time analysis, polyvariant mixed computation, and post-processing optimizations. We describe each of these phases below.

2.1 Binding-Time Analysis

Initially, the partial evaluator is given the names of the basic functions of the ealgebra which will be known ahead of time. During *binding-time analysis*, the partial evaluator determines which basic functions can be pre-computed in the next phase. This process is called binding-time analysis because it determines at what time the value(s) of a basic function can be determined (*i.e.* bound to known values).

The input to this phase is a division of the basic functions which supply input to the ealgebra into two sets: *positive* functions whose values will be known ahead of time, and *negative* functions whose values will not be known until later. The partial evaluator proceeds to classify all basic functions (including those not initially marked by the user) as positive or negative. ([3] use the terms "static" and "dynamic" to refer to these types of values; these terms have different meanings within the ealgebra paradigm.)

In the current implementation, the following algorithm is used to classify a function f as positive or negative:

- If f is syntactically static (that is, not updated by any transition rule), f remains as classified initially by the user.
- If an update $f(\bar{t}) := t_0$ exists in p such that $\bar{t}$ or t_0 references a negative function, f is negative. (Note that even if f was declared as positive by the user, f may still depend on other negative functions and must be classified as negative.)
- If for all updates $f(\bar{t}) := t_0$ in p, every function referenced in $\bar{t}$ and t_0 is positive, and f is not already negative, f is positive.

This classification algorithm is repeatedly applied until a fixed-point is reached. Any remaining unclassified functions are classified as negative and the algorithm is repeated to ensure consistency.

An interesting problem which this algorithm does not handle is the problem of circular dependencies. A basic function f is *self-referential* if some update $f(\bar{t}) := t_0$ within the ealgebra being specialized contains a reference to f within $\bar{t}$ or t_0. The above algorithm classifiess every self-referential function as negative. Often this is appropriate, as some self-referential functions can grow unboundedly. But at times, classifying such functions as positive is also appropriate. Consider the following program:

if *Num > 0* **then** *Num := Num + 1* **endif**
if *MyList* $\neq$ *Nil* **then** *MyList := Tail(MyList)* **endif**

Suppose the initial values of *Num* and *MyList* are known. *Num* should not be classified as a positive function, since it would lead the specializer in the next stage into an infinite loop, as larger and larger values of *Num* would be computed as positive information. On the other hand, there is no problem with classifying *MyList* as positive, since *MyList* will eventually be reduced to *Nil* and remain at that value forever. An addition to the algorithm presented above properly classifies self-referential functions as positive if they are dependent only upon themselves in a bounded manner (as seen here).

The problem of circular dependencies is much more general that the problem of self-reference; it may be that several functions form a mutual dependency cycle. We intend to incorporate a more sophisticated algorithm for binding-time analysis based on an examination of the dependency graph formed by the basic functions of the algebra. In the future, we hope to extend this analysis to parts of basic functions; it may be that $f(\bar{t})$ could be classified as positive for certain tuples $\bar{t}$ but not for others.

2.2 Polyvariant Mixed Computation

After binding time analysis, the partial evaluator begins the process of specializing the input program, executing rules which depend only on positive information (that is, functions classified as positive by our binding-time analysis) and generating code for rules which depend on negative information. The process is called *polyvariant mixed computation*: "mixed" because the processes of executing positive rules and

generating code for negative rules is interleaved, and "polyvariant" because the entire program is considered multiple times for different sets of positive information.

The signature τ of the algebra has been divided into two components during binding time analysis: a positive signature τ_+ and a negative signature τ_-. This leads to a corresponding division of states (or structures) S into structures S_+ and S_-. The partial evaluator creates an ealgebra with signature $\tau_- \cup \{K\}$, where K is a nullary function which will be used to hold the positive (or "known") information formerly stored by functions in τ_+.

From a given positive state S_+, the partial evaluator produces rules of the form

$$\textbf{if } K = S_+ \textbf{ then } \textit{rules} \textbf{ endif}$$

where *rules* is a specialized version of the rules of the entire input program with respect to S_+, along with an assignment to K. Call a transition rule of this form a *K-rule*, whose *guard* is *($K = S_+$)* and whose *body* is *(rules)*. Not that no two K-rules produced by our partial evaluator will have the same guard. We recursively describe how transition rules are speciailized with respect to a given positive state S_+ below.

An expression is specialized with respect to S_+ by substituting all known values of functions in S_+ into the given expression, simplifying when possible.

An update $f(\bar{t}) := t_0$ is specialized with respect to S_+ as follows. If $f \in \tau_+$, no rule is generated. Instead, the change to the positive function f is noted internally in order to generate the correct assignment to K. (Note that in this case, all functions named in $\bar{t}$ and t_0 are positive as a result of our binding time analysis.) Otherwise, an assignment to f is generated, with the values of t_0 and $\bar{t}$ specialized as much as possible using the information in S_+.

A set of rules is specialized with respect to S_+ by specializing each rule and combining the information needed to create a single assignment to K.

A guarded rule "**if** *guard* **then** R_1 **else** R_2 **endif**" is specialized with respect to S_+ as follows. If all functions in *guard* are positive, the result is the specialization of R_1 or R_2, depending on whether the value of *guard* is true or false in S_+. Otherwise, an **if** statement is generated, with *guard*, R_1, and R_2 specialized as above. A guarded rule containing **elseif** clauses is converted to an equivalent form without **elseif** clauses and specialized as above.

2.3 Optimization

The above transformations create a specialized version of the original program. Often, this specialized version contains many unneeded rules, such as:

$$\textbf{if } K = \textit{foo} \textbf{ then } K := \textit{bar} \textbf{ endif}$$

Such a K-rule can be deleted; just replace all references to *foo* in the program with references to *bar*. The partial evaluator performs several such optimizations on the specialized program:

- Eliminating terms which serve only as aliases for other terms or constants.
- Combining K-rules with identical bodies.
- Combining K-rules which are executed consecutively but whose bodies have independent updates that could be executed simultaneously without altering the meaning of the program.
- Eliminating K-rules which will never be executed.

These optimizations generate code which is equivalent, but textually shorter and usually requires fewer moves to execute. Of course, one pays a small price in time in order to generate these optimizations.

2.4 Results

It is important that a partial evaluator actually perform useful work. Kleene's s-m-n theorem shows that partial evaluators can in principle be constructed; his proof shows that such evaluators may not necessarily produce output that is more efficient than the original. One can specialize an ealgebra, for example, by creating two K-rules: one which initializes the functions in S_+ to their initial values and one which has the original unspecialized program as its body. This "specialized" ealgebra has the same behavior as the original, but is hardly more useful than the original algebra.

[3] suggest a standard for evaluating partial evaluators. Consider a self-interpreter for a language: that is, an interpreter for a language L written itself in L. (McCarthy's original description of a LISP interpreter written itself in LISP is such an interpreter.) Specializing such an interpreter with respect to an L-program p should yield a version of p, of size comparable to p, as output. That is, the overhead involved in interpreting a language (deciding which command is to be executed, incrementing a program counter, *etc.*) should be removed by a good partial evaluator. Our partial evaluator seems to approach this standard when run on small programs, though more detailed testing is needed.

3 An Example

Consider the C function `strcpy`:

```
void strcpy (char *s, char *t) { while (*s++ = *t++) ; }
```

This function copies a string from the memory location indicated by t to the memory location indicated by s. It is admittedly cryptic.

In [2], we presented an ealgebra interpreter for the C programming language. As a test, we ran our partial evaluator on our algebra for C, specializing it with respect to `strcpy()`. The result, with most of the functions renamed, appears below.

if *K = "init"* **then** *CopyFrom :=* `t`*, CopyTo :=* `s`*, K := "first-update"* **endif**
if *K = "first-update"* **then**
    TmpFrom := CopyFrom, TmpTo := CopyTo
    CopyFrom := CopyFrom + 1, CopyTo := CopyTo + 1, K := "loop"
endif
if *K = "loop"* **then**
    if *Memory(TmpFrom) ≠ 0* **then**
        Memory(TmpTo) := Memory(TmpFrom)
        TmpFrom := CopyFrom, TmpTo := CopyTo
        CopyFrom := CopyFrom + 1, CopyTo := CopyTo + 1, K := "loop"
    else *Memory(TmpTo) := Memory(TmpFrom), K := "done"*
    endif
endif

This algebra is considerably smaller than the entire ealgebra for C, and hopefully is more easily understandable than the original C code. It is not optimal: for example, *CopyFrom* could be replaced by `t`, since `t` is never used after the initial state. It does, however, exhibit the behavior of `strcpy()` more directly than the entire ealgebra for C given `strcpy` as input.

(For those familiar with [2], the input functions initially specified as positive were *CurTask*, *TaskType*, *NextTask*, *LeftTask*, *RightTask*, *TrueTask*, *FalseTask*, *Decl*, *WhichChild*, and *ChooseChild*, assuming that *ChooseChild* always moves to the left first.)

References

[1] Yuri Gurevich, "Evolving Algebras 1993: Lipari Guide", in *Specification and Validation Methods*, ed. E. Börger, Oxford University Press, to appear.

[2] Yuri Gurevich and James K. Huggins, "The Semantics of the C Programming Language", Selected papers from CSL'92 (Computer Science Logic), Springer Lecture Notes in Computer Science 702, 1993, 274–308.

[3] N.D. Jones, C.K. Gomard, and P. Sestoft, *Partial Evaluation and Automatic Program Generation*, Prentice Hall, 1993.